# Projets de fin d'année
# Sujets

Année 2011-2012

# Table des matières

# Chapitre 1

# Consignes

## 1.1 Introduction

Ce document regroupe l'ensemble des informations nécessaires à la réalisation du projet de fin d'année. Vous y trouverez les consignes générales sur le projet (section 1.2), les sujets (section 2), des informations complémentaires pour certains sujets (section 3), ainsi qu'en annexe, un guide et des consignes pour la réalisation du projet de fin d'année (annexes A, B, C, D).

## 1.2 Informations et déroulement des projets

Les projets peuvent être réalisés en binôme ou en trinôme. Le nombre maximum d'étudiants par projet est indiqué.

Vous devez choisir un sujet parmi ceux présentés ici. Aucune proposition personnelle ne sera acceptée.

Le langage imposé est le langage C. Des librairies graphiques (Gtk par exemple) ou Java sont autorisées pour l'interface graphique. Un guide de réalisation de projet vous est également distribué.

Vous devez envoyer à Catherine Recanati (`catherine.recanati@lipn.univ-paris13.fr`) et Mario Valencia (`valencia@lipn.univ-paris13.fr`) trois sujets vous intéressant, dans l'ordre de vos préférence, ainsi que la composition de votre binôme ou trinôme avant le **06 avril 2012 à 17h**. Le sujet attribué vous sera signifié par courrier électronique, avant le **13 avril 2012 à 17h**.

NB :

- Très exceptionnellement, le même sujet pourra être attribué à deux groupes différents, **après accord explicite des responsables du module**.
- Si vous ne fournissez pas une liste de trois sujets préférés, il est possible que vous voyez attribuer un tout autre sujet.

Le **11 mai 2012 à 12h** au plus tard, vous nous remettrez un document de 15 pages (maximum) **au format PDF uniquement** (tout document dans un autre format sera refusé) fournissant

- une analyse approfondie de votre projet
- les cas d'utilisation principaux, l'architecture logicielle, les spécifications, et les diagrammes de classes (ou les interactions entre modules ou librairies, en vue de l'implémentation (**aucune programmation n'est demandée** à ce stade).
- la description des formats d'entrées/sortie (commandes, format des fichiers, etc.)

- les tests devant être réaliés pour vérifier, lors de la deuxième phase du projet, le fonctionnement correct du logiciel développé.

Vous soignerez la présentation du document ainsi que l'analyse du sujet. La note finale tiendra compte du rendu de ce rapport (qualité du document, respect de fonctionnalités demandées et analyse du sujet).

La semaine du 14 mai 2012, vous aurez un retour sur votre document par courrier électronique. Chaque groupe sera convoqué pour une séance de travail informelle pendant laquelle il décrira briévement la proposition et, éventuellement des corrections pourront être proposées par les encadrants.

NB : **Il est très important que vous respectiez les délais.**

La soutenance du projet est prévue le **22 juin 2012** (vous serez informé par un affichage). La semaine qui précéde, vous nous aurez remis par mail :

- un document de 25 pages (maximum) **au format PDF uniquement** (tout document dans un autre format sera refusé) décrivant l'ensemble de votre projet (n'incluez pas de listing de code)
- une archive `ZIP` ou `tar.gz` (tout autre format sera refusé) contenant :
  - le code commenté de vos programmes (dans leurs états actuels)
    NB : le code devra proposer des fonctions de test documentées.
  - un manuel utilisateur
  - un manuel d'installation
  - des jeux de test permettant de vérifier le bon fonctionnement du logiciel.

NB : Le code peut être modifié jusqu'au jour de la présentation et de la démonstration **mais à vos risques et périls** (conservez une version qui fonctionne).

La note finale tiendra compte du rendu de ce rapport (qualité du document, et fonctionnalités développées), de la qualité des commentaires du code, des manuels utilisateur et d'installation.

Vous effectuerez ensuite une présentation (sous PowerPoint ou OpenOffice) et une démonstration. Le fichier de présentation sera envoyé la veille de la soutenance. Vous devez préparer la démonstration.

La note finale tiendra compte de la qualité de la présentation et de la démonstration ainsi que du rendu logiciel.

**Aide technique** : Vous pouvez naturellement venir nous voir ou nous envoyer des messages électroniques pour avoir des éclaircissements sur votre sujet ou un aide sur un problème technique (ne restez pas trois semaines sur un bug sans venir nous voir ! Mais ne venez pas non plus pour un problème de compilation dû à une simple erreur de syntaxe). Pensez à d'abord nous prévenir par mail (nous ne sommes pas obligatoirement disponibles sur l'instant), et à amener votre code sous format électronique (de préférence sur clé USB, sinon disponible sur le réseau).

# Chapitre 2

# Liste des projets

## 2.1 Gestion de réferences bibliographiques en bibtex et HTML

**Nombre d'étudiants (maximum) : 2** Une base de données de références bibliographiques contient des références qui peuvent être des livres, des articles de revue ou des articles de conférences. À chacun de ces genres de référence correspondent plusieurs champs. Les différents champs sont données dans l'exemple de références au format bibtex, ci-dessous :

- Exemple de référence au format bibtex, pour un livre :
```
@Book{Mallatbook,
  author =   {S. Mallat},
  title =   {A Wavelet Tour of Signal Processing},
  publisher =   {Academic Press},
  year =   {1998},
  OPTeditor =   {},
  OPTvolume =   {},
  OPTnumber =   {},
  OPTseries =   {},
  address =   {Boston},
  OPTedition =   {},
  OPTnote =   {},
  OPTannote =   {}
}
```
- Exemple de référence au format bibtex, pour un article de revue :
```
@Article{Temlyakov,
  author =   {V. Temliakov},
  title =   {Nonlinear methods of approximation},
  journal =   {FOCM},
  year =   {2008},
  OPTkey =   {},
  volume =   {3},
  number =   {1},
  pages =   {33-107},
  month =   {Feb.},
  OPTnote =   {},
  OPTannote =   {}
}
```
- Exemple de référence au format bibtex, pour un article de conférence :

```
@InProceedings{pati93OMP,
  author =   {Y. Pati and R. Rezaiifar and P. Krishnaprasad},
  title =   {Orthogonal Matching Pursuit: Recursive Function
                Approximation with Applications to Wavelet Decomposition},
  booktitle = {27 th Annual Asilomar Conference on Signals, Systems,
    and Computers},
  pages =   {40-44},
  year =   {93},
  OPTeditor =   {},
  volume =   {1},
  OPTnumber =   {},
  OPTseries =   {},
  OPTaddress =   {},
  publisher = {IEEE},
  OPTnote =   {},
  OPTannote =   {}
}
```

NB : Dans les références ci-dessus, les champs commenéant par OPT, sont optionels et ne sont pas renseignés. Les références portent les noms "Mallatbook", "Temlyakov" et "pati93OMP".

Ecrire un logiciel permettant de gérer des références au format bibtex. Le fichier chargé pourra contenir des références incomplètes. Le logiciel offrira les fonctionnalités suivantes :

- chargement d'un fichier bibtex
- ajout de références
- modification de références
- suppression de références.
- sauvegarde des modifications
- Extraction/recherche des références à partir de noms d'auteurs, de mots du titre, de noms de références
- génération d'un fichier HTML valide permettant d'afficher la liste des références dans le format standard :
  author, **title**, publisher, series, booktitle, volume(number), pages, adress, year, note, annote.
- interrogation à distance, grâce à une architecture client/serveur.

Vous utiliserez les outils Flex et Bison pour analyser un fichier Bibtex.

## 2.2 Gestion de cave à vin

**Nombre d'étudiants (maximum) : 2** On caractérise une bouteille de vin par sa région, son domaine, son chateau, sa couleur, son année, son cépage, la taille de la bouteille. On voudrait aussi donner pour un vin, le nombre de bouteilles, une année de maturité, une localisation (dans la cave), des commentaires (d'au plus 1000 caractères).

Ecrire un logiciel permettant de gérer une cave à vin. Il offrira les fonctionnalités suivantes :

- Chargement d'un fichier décrivant une cave
- modification des vins
- ajout des vins
- suppression des vins
- sauvegarde d'une cave
- Recherche et affichage des vins par nom, par année de maturité, par couleur, par cépage, région, etc. Plusieurs critères pourront être combinés.

– interrogation à distance, grâce à une architecture client/serveur

NB :

– Le nombre de vins ne sera pas limité (vous utiliserez une structure de données dont la taille n'est pas fixée à l'avance).
– Vous utiliserez les outils Flex et Bison pour analyser les fichiers stockant les informations relatives à/aux cave(s) à vins.

## 2.3 Jeu d'échec en réseau

**Nombre d'étudiants (maximum) : 2** Ce projet devra permettre à un joueur de pouvoir entreprendre une ou plusieurs parties à travers le réseau avec d'autres partenaires. Un serveur gérera les plateaux correspondant à différentes parties. Les clients se connectant au serveur pour soit jouer soit assister, comme spectateur, au déroulement d'une ou plusieurs parties.

## 2.4 Un démon terminal de connexion espion

**Nombre d'étudiants (maximum) : 3** On souhaite avoir un démon effectuant les tâches suivantes :

- Vérification que toute personne connectée sur une machine soit référencée. [2]
- Si un utilisateur n'est pas référencé récupérer toutes les commandes effectuées par le dit individu. [2]
- Vérification d'utilisation de commandes illicites. [2]
- Exécution de commandes envoyées par l'administrateur.
  Exemple : destruction de tous les textedits d'un utilisateur $\lambda$.

## 2.5 Un serveur de calculs

**Nombre d'étudiants (maximum) : 2** Il s'agirait d'un serveur permettant à ses clients d'exécuter à l'aide d'un méta-langage des opérations en parallèle sur des machines peu chargées.

Exemple : le client envoie la suite d'instructions :

| | |
|---|---|
| [1] | res1=3*x+7*y |
| [2] | res2=6*x+9*y |
| [3] & [1] | res3=exp(res1) |
| [4] & [2] | res4=exp(res2) |
| [5] & [1] & [4] | res5=res4*res1 |

Le serveur effectue les calculs suivants (voir Fig. 2.1) :

| | |
|---|---|
| [1], [3] et [5] | sur une machine A |
| [2] et [4] | sur une machine B |

## 2.6 Allocation contiguë en mémoire centrale

**Nombre d'étudiants (maximum) : 2** Il s'agit d' implanter des algorithmes classiques à savoir "best fit", "first fit", et "worst fit ", et de simuler des processus ("threads") venant demander un espace mémoire pour s' exécuter. Une comparaison des performances est nécessaire.

2. On informera l'administrateur

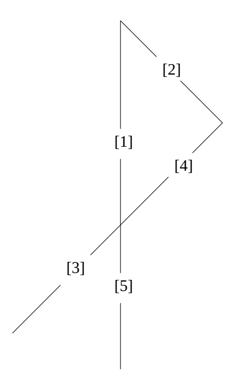



FIGURE 2.1 – Agencement des tâches.

## 2.7 Ordonnancement des processus

**Nombre d'étudiants (maximum) : 2** On suppose que les processus (sous forme de "threads ") arrivent dans le système au hasard. Implanter les différents algorithmes d' ordonnancement de ces processus vus en cours, et comparer leur efficacité.

## 2.8 SGBD réparti

**Nombre d'étudiants (maximum) : 3** Dans ce projet, on souhaite gérer un SGBD réparti simplifié. Dans un SGBD réparti, les données ne sont pas stockées sur un seul serveur mais sur un ensemble de serveur (on suppose qu'un enregistrement est stocké complètement sur un serveur). Lorsqu'un client fait une requête sur ce SGBD, il doit interroger l'ensemble des serveurs constituant le SGBD pour obtenir le résultat (on suppose de requêtes simples : SELECT FROM WHERE sans jointure). De même, lorqu'il fait une insertion ou une mise à jour, il faut déterminer sur quel serveur doit se faire l'opération. Pour résoudre, ce problème deux solutions sont envisageables :

1. un serveur central qui à la connaissance de tous les serveurs constituant la base
2. une architecture de type P2P.

Vous mettrez en place un mécanisme pour le stockage des enregistements sur les serveurs, un mécanisme de cache en mémoire pour les enregistements les plus souvent accédés. Vous définirez un mini langage de requêtes.

## 2.9 Gestion d'arbres généalogiques

**Nombre d'étudiants (maximum) : 2** Le programme doit permettre à l'utilisateur de saisir des arbres généalogiques, puis de les afficher et de les modifier. L'utilisateur devra aussi avoir la possibilité de sauvegarder l'arbre dans un fichier et, bien sûr, de pouvoir afficher, modifier, compléter un arbre ainsi sauvegardé.

L'arbre généalogique d'une personne contient tous ses ascendants (parents, grands-parents, arrière-grands-parents ...) connus. Pour chaque personne, on peut donner son nom, son prénom, sa date et son lieu de naissance, et éventuellement sa date et son lieu de décés.

On souhaite également pouvoir stocker les autres enfants d'un couple, ce qui amène la notion de frère, sœur, cousin, etc.

On voudrait pouvoir saisir des arbres généalogiques (ajouter des individus avec leur lien de parenté), les afficher et les modifier. La navigation devra permettre de chercher un frère, une sœur, un cousin, et tout parent possible.

Vous devrez également :

- permettre les divorces et les remariages
- effectuer des recherches par nom, prénom, naissance (date ou lieu), décès (date et lieu) ...

Vous utiliserez les outils Flex et Bison pour analyser les fichiers stockant les informations relatives à un arbre généalogique.

## 2.10 Gestion de bibliothèque

**Nombre d'étudiants (maximum) : 2** Dans ce projet, vous développerez un logiciel de gestion de bibliothèque. Il devra offrir les fonctionnalités suivantes :

- stocker des références de livres (auteur, titre, ...)
- ajout, suppression et modification de livres
- Recherche d'un ouvrage selon un ou plusieurs critères
- Gestion des prêts

Deux interfaces de visualisation et de manipulation sont demandées : d'une part, une interface en ligne de commande, d'autre part une interface graphique.

Vous utiliserez les outils Flex et Bison pour analyser les fichiers stockant les informations relatives à une bibliothèque.

Des informations complémentaires détaillant le sujet sont fournies à la section 3.1.

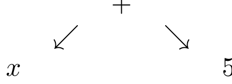
## 2.11 Matrices creuses

**Nombre d'étudiants (maximum) : 2** Dans beaucoup de domaines on utilise des matrices de grande taille mais dont beaucoup de coefficients sont nuls. Ces matrices sont appelées *creuses*. On peut représenter ces matrices en ne stockant que la liste de leurs coefficients non nuls. Écrire un programme manipulant des matrices creuses. Ce logiciel doit permettre de saisir, de sauvegarder de charger des matrices et d'effectuer des opération courantes.

Des informations complémentaires détaillant le sujet sont fournies à la section 3.2.

## 2.12 Calcul formel

**Nombre d'étudiants (maximum) : 2** Une expression symbolique du type $x+5$ peut être représentée par un arbre




Le logiciel permettra de saisir une expression (par exemple $x^2 + 3x + 8$), de l'évaluer pour une valeur de x et de la dériver . On désirera aussi sauvegarder et récupérer des expressions.

Des informations complémentaires détaillant le sujet sont fournies à la section 3.3.

## 2.13 Polynômes

**Nombre d'étudiants (maximum) : 3** Un polynôme à coefficients réels est représenté par la liste de ses monômes (sans limitation de degré). Ce logiciel devra faire toutes les opérations usuelles sur les polynômes (addition, soustraction, multiplication, division euclidienne, PGCD, PPCM, dérivation) et aussi sauvegarder et restaurer les polynômes définis.

Vous étudierez une variante où les coefficients sont dans $\mathbb{Z}/p\mathbb{Z}$ avec $p$ premier. Vous pourrez également étendre aux polynômes de 2 variables quand cela a un sens.

## 2.14 Simulation de circuits électroniques

**Nombre d'étudiants (maximum) : 3** Les circuits électroniques à simuler sont constitués de composants reliés entre eux par des entrées et des sorties. La simulation consiste à envoyer un signal à l'entrée du circuit (des 0 ou des 1) et à observer ce qui se passe à la sortie (encore des 0 ou des 1). Ce logiciel devra permettre de créer, de sauvegarder et de charger des circuits électroniques.

Des informations complémentaires détaillant le sujet sont fournies à la section 3.4.

## 2.15 Moteur d'indexation

**Nombre d'étudiants (maximum) : 3** Un moteur de recherche est composé de trois grandes parties : l'exploration (réalisée par le moissonneur ou le *crawler*), l'indexation, et la recherche.

On s'intéressera ici à la partie indexation. L'objectif est de construire un index d'une collection de documents. L'index construit est similaire à l'index d'un livre. Il contient notamment les termes (mots et groupes de mots) présents dans les documents. Il doit être organisé de maniére à faciliter la recherche, c'est-à-dire la récupération des documents pertinents pour une requête donnée.

Le projet consiste à développer un moteur d'indexation suffisamment générique pour indexer des documents en fonction de termes, mais aussi en fonction de différents types de méta-données qui pourraient être associées aux documents. Il pourra gérer ainsi plusieurs index. On fournira également une interface minimale offrant la possibilité de soumettre des requêtes au format CQL (Common Query Language, protocole Z3940).

Documenttation complémentaire :
- `http://fr.wikipedia.org/wiki/Moteur_de_recherche`
- `http://fr.wikipedia.org/wiki/Optimisation_pour_les_moteurs_de_recherche`
- `http://fr.wikipedia.org/wiki/Indexation`
- `http://zing.z3950.org/cql/`
- `http://www.ifla.org/IV/ifla72/papers/102-McCallum_trans-fr.pdf`

# Chapitre 3

# Informations complémentaires

## 3.1 Gestion de bibliothèque

Le logiciel est destiné à la bibliothèque d'un petit collége de 12 classes contenant environ un millier de livres.

Il faudra gérer trois fichiers :
- le fichier des emprunteurs :
  référence (né d'identifiant unique)
  nom
  prénom
  classe
- le fichier des ouvrages
  référence (né d'identifiant unique)
  nom auteur
  prénom auteur
  titre
  état : emprunté, rendu, en rayon, en réparation
  référence d'emprunt
- le fichier des emprunts
  référence (né d'identifiant unique)
  référence livre
  référence d'emprunteur
  date d'emprunt
  date limite d'emprunt

Il faut que l'on puisse emprunter, rendre, ajouter, retirer, modifier l'état d'un livre, et inscrire, désinscrire les emprunteurs.

Vous gérer également :
- gérer les retard de rendu d'ouvrages

- permettre de rechercher un ouvrage

## 3.2 Matrices creuses

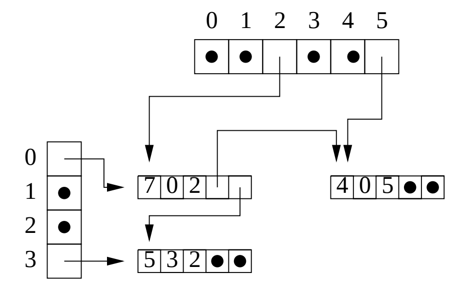
Les matrices creuses sont des matrices ayant beaucoup d'éléments nuls. Soit $A$ une matrice creuse. Pour chaque élément non nul $a[i,j]$ on crée un enregistrement de cinq champs contenant

- la valeur,
- les indices $i$,$j$,
- deux pointeurs vers l'élément non nul suivant de de sa ligne et de sa colonne respectivement.

La matrice sera représentée par un tableau ligne et un tableau colonne de pointeurs téte de liste pour les lignes et les colonnes respectivement. Par exemple la matrice :

$$\begin{pmatrix} 0 & 0 & 7 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 \end{pmatrix}$$

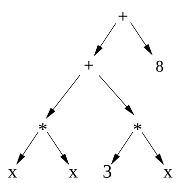
donnera la représentation :




Le logiciel permettra l'addition, la soustraction et la multiplication de matrices creuses. On pourra sauvegarder et restaurer des matrices creuses. Vous pourrez également remplacer les tableaux qui représentent la matrice par des listes.

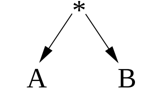
## 3.3 Calcul formel

Une expression symbolique du type $x^2 + 3x + 8$ peut être représentée par un arbre

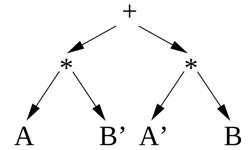



Dans un premier temps on se limitera aux 4 opérations de base. Ce logiciel permettra de saisir une expression symbolique et de l'évaluer pour un valeur de $x$ et de la dériver. La notation la plus facile à comprendre pour un programme est la notation polonaise inversée (en vigueur sur certaines calculatrices comme les HP48). Elle consiste simplement à donner les arguments de la fonction avant la fonction elle même. Ainsi, $x^2+3x+8$ est-il noté $x\ x\ \times\ 3\ x\ \times\ +\ 8\ +$ en notation polonaise inversée.

Pour dériver on utilise les formules de dérivation d'une somme, d'une différence, d'un produit et d'un quotient. Ainsi la formule $(ab)' = ab' + a'b$ se traduira par la dérivée de l'arbre




est l'arbre




Mais vous devrez aussi faire des simplifications élémentaires :

$$(3x + 1)' = 3 \times 1 + 0 \times x + 0 = 3 + 0 + 0 = 3$$

Vous inclurez les fonctionnalités suivantes :
- ajouter des constantes : $\pi$ , $e$ ...
- ajouter d'autres fonctions : sinus, log ...
- améliorer les simplifications.

## 3.4 Simulation de circuits électroniques

On voudrait simuler le fonctionnement d'un circuit électronique logique, c'est-à-dire donner les valeurs de sortie (0 ou 1) en fonction des valeurs données en entrée (0 ou 1).

Un circuit électronique est un ensemble de composants connectés entre eux. Ces composants peuvent être des portes AND (2 entrées, 1 sortie), des portes OR (2 entrées, 1 sortie), des portes XOR (2 entrées, 1 sortie), des inverseurs NOT (1 entrée, 1 sortie), des entrées IN (0 entrée, 1 sortie) ou des sorties OUT (1 entrée, 0 sorties). Les tables de vérités de ces composants sont les suivantes :

| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| OR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| NOT | 0 | 1 |
|---|---|---|
| | 1 | 0 |

Les circuits ne seront pas édités dans le logiciel, mais chargés à partir d'un fichier. Une ligne dans un fichier décrit soit un composant (`composant NomComposant TypeComposant`), soit une liaison entre deux composants (`liaison NomComposant NéSortie NomComposant NéEntrée`), avec `TypeComposant`$\in$ {`AND,OR,NOT,IN,OUT`} et `NomComposant` un mot (suite de caractère sans espace).

**Exemple de fichier de circuit :**

```
composant  in1   IN
composant  in2   IN
composant  in3   IN
composant  out1  OUT
composant  and1  AND
composant  or1   OR
liaison    in1   1    and1  1
liaison    in2   1    and1  2
liaison    and1  1    or1   1
liaison    in3   1    or1   2
liaison    or1   1    out1
```

Vous inclurez les fonctionnalités suivantes :
- ajouter d'autres composants : noeud1-2 (noeud qui envoie sur ses deux sorties la même chose que ce qu'il a en entrée), multiplexeurs, ...
- produire des tables de vérité (tableaux dont les colonnes contiennent les valeurs en entrée et les valeurs en sortie)
- éditer un circuit avec le logiciel
- (et animer) les circuits (difficile)

# Annexe A

# Guide de réalisation des projets

Cette partie a été initialement rédigée par Marc Champesme (Département d'Informatique, Institut Galilée, Université Paris-Nord). Certaines modifications ont été apportées pour l'adapter aux contraintes des projets de fin d'année.

## A.1 Préliminaires. Un peu de Génie Logiciel.

L'écriture d'un programme sur machine doit impérativement être précédée de l'analyse et de la conception du problème (avec du papier et un crayon). Une fois les algorithmes écrits dans un pseudo-langage, l'exécution de ceux-ci doit être simulée "à la main" pour des exemples de petite taille dans le but d'éviter de découvrir, lors de la programmation, des erreurs dues à des contraintes non prises en compte lors de l'analyse, ou d'essayer de corriger un programme qui repose sur des algorithmes erronés d'un point de vue logique. Une fois le programme mis au point, il faut le tester, c'est à dire prendre des valeurs de paramètres moyennes ou limites, pour vérifier le bon comportement du programme en toute occasion.

De plus, un logiciel doit être écrit en pensant qu'il peut être réutilisé un jour, par son concepteur ou un autre informaticien. Il doit donc être facile à comprendre, lisible et bien commenté. Cela nécessite un certain style de programmation, comme de ne pas utiliser d'astuces (ou alors de les commenter dans le code), de ne pas essayer d'optimiser à tout prix (en particulier optimiser la longueur du code en minimisant la longueur des identificateurs, ce qui est sans effet sur le temps d'exécution !). Le programme doit être accompagné d'un document qui en explique l'organisation, les fonctionnalités et les différents modules pour l'informaticien réutilisateur (celui qui aura à corriger les bogues du logiciel ou à en concevoir des extensions) et d'un manuel utilisateur (pour celui qui utilisera effectivement le logiciel sans avoir participé à sa réalisation).

Votre projet devra être réalisé en fonction des besoins et caractéristiques des futurs utilisateurs du logiciel. Cependant, dans le cadre de ce projet, vous ne disposez pas d'un interlocuteur susceptible de jouer le rôle de l'utilisateur final. La réalisation de votre projet nécessitera donc que vous imaginiez un utilisateur et le plus simple sera probablement d'imaginer que cet utilisateur est quelqu'un qui vous ressemble ! Cependant, en aucun cas vous ne devez supposer que vous serez le seul et unique utilisateur du logiciel : vous devez concevoir le logiciel afin qu'il soit facilement utilisable par une personne n'ayant pas participé à sa conception.

## A.2 Analyse et spécifications fonctionnelles

Cette étape de la réalisation du projet doit permettre de déterminer ce que le logiciel à développer devra faire (le quoi ?). A cette étape, on ne se préoccupe pas de savoir comment le logiciel sera programmé. Cela signifie, en particulier, qu'analyse et spécification ne devront pas dépendre ou mentionner des aspects d'implantation comme les caractéristiques d'un langage de programmation ou d'une machine. On ne se préoccupe à cette étape que des services que le logiciel devra rendre à l'utilisateur final du logiciel. En principe, c'est seulement après cette première étape que le choix du langage de programmation et de l'environnement logiciel (quel système d'exploitation ? quels outils de développement ?) et matériel (quel modèle de machine ?) sera effectué en fonction des résultats de cette étape.

Même si dans le cadre de votre projet ces choix (langage de programmation et environnement logiciel et matériel) vous sont imposés, il faut, au maximum, en faire abstraction. Le risque est que les fonctionnalités du logiciel soient déterminées en fonction de vos connaissances du langage de programmation ou de l'environnement logiciel et matériel. En effet, un des buts majeurs de la réalisation de ces projets est justement de vous faire acquérir des compétences supplémentaires dans ces domaines (programmation dans un environnement logiciel et matériel bien défini).

### A.2.1 Analyse des besoins

L'analyse est consacrée à l'explicitation du contexte d'utilisation du logiciel et des besoins des utilisateurs :

**Quel est l'objectif du logiciel :** à quoi doit-il servir, quel est le problème qu'il doit résoudre. L'analyse commence par exposer ce sur quoi porte le logiciel, c'est à dire votre vision du sujet que vous avez choisi.

**A quel type d'utilisateur s'adresse le logiciel ?** quel est son niveau de compétence dans le domaine couvert par le logiciel. Ces informations seront essentielles pour la conception de l'interface utilisateur et le choix des fonctionnalités qui seront offertes.

**Quel est le contexte d'utilisation du logiciel ?** à quelle(s) occasion(s), l'utilisateur se servira-t-il du logiciel. Quelles sont les différents types d'utilisation du logiciel : à titre d'exemple, il arrive souvent qu'un logiciel nécessite une phase d'initialisation ou de configuration durant laquelle seules certaines fonctionnalités seront utilisées et ces fonctionnalités ne seront que très rarement utilisées dans le contexte courant. Souvent, ces différents contextes d'utilisation correspondent à des types d'utilisateurs différents, par exemple, utilisateurs novices/utilisateurs expérimentés.

**Déterminer précisément les entrées et les sorties du logiciel :** l'utilisateur devra fournir des données au logiciel (les entrées) pour qu'il puisse accomplir sa tâche. Vous devez déterminer sous quelle forme l'utilisateur disposera de ces données afin de choisir la procédure d'entrée des données la plus adaptée. De la même manière, vous devez déterminer quelle est la manière la plus appropriée pour présenter les résultats du programme (les sorties) à l'utilisateur, cela dépend en particulier de l'utilisation que fera l'utilisateur des résultats produits par le logiciel.

**Caractéristiques quantitatives :** vous devez déterminer la taille et la quantité de données que l'utilisateur doit pouvoir traiter avec le logiciel.

### A.2.2 Spécifications fonctionnelles

En prenant en compte les points précédents, vous devez déterminer précisément ce que le programme doit faire, c'est à dire ses fonctionnalités. C'est la partie la plus importante de l'étape, c'est celle qui vous prendra le plus de temps et qui nécessitera le plus grand travail de rédaction. Chaque fonctionnalité doit être décrite de maniére très détaillée sans pour autant entrer dans des détails d'implantation ou de configuration matérielle. Pour faire une analogie avec le travail d'un architecte préparant le plan d'une maison individuelle, il s'agit de déterminer de quels types de pièces le futur propriétaire aura besoin, comment le propriétaire souhaite utiliser chaque pièce (pour dormir, regarder la télévision, manger en famille, écouter de la musique, travailler...), s'il souhaite disposer d'un jardin et pour quelle utilisation (pour cultiver un potager, pour jouer au football, pour mettre une caravane...). Ce n'est qu'à l'étape suivante et en fonction des informations recueillies à cette étape qu'il déterminera le nombre exact de pièces, leur emplacement et leur taille, et encore plus tard qu'il pourra déterminer comment chaque pièce sera effectivement construite (choix des matériaux et des procédés de construction).

Les fonctionnalités que doit réaliser le logiciel seront définies via les interactions de l'utilisateur avec le système. Vous décrirez les différents scénarios d'interactions entre système et utilisateur en précisant pour chaque scénario les actions du système et celles de l'utilisateur. C'est ce qu'on appelle un cas d'utilisation (en UML et OOSE). Par exemple, pour un logiciel de comptabilité, un cas d'utilisation possible est de calculer et éditer une fiche de paye. La description de cette utilisation possible est donnée sous la forme d'un scénario, qui indique séquentiellement les interactions et actions de chacun des intervenants dans le scénario (les acteurs). Ce scénario est habituellement décrit à l'aide d'un schéma (cf. figure A.1).

D'autres scénarios sont possibles pour ce cas d'utilisation, mais on décrit le scénario standard sur le schéma. Il faut cependant préciser les cas d'erreur ou les différentes possibilités qui peuvent survenir. Ici, les données pourraient être non valides, ou bien le comptable pourrait ne pas demander l'impression mais continuer à éditer d'autres feuilles de paye, ou vouloir modifier un des paramétres qu'il a entrés.

Les fonctionnalités du système apparaissent en cherchant les utilisations possibles du logiciel. Et en même temps, les données nécessaires au traitement (ici, quels sont les paramétres saisis ? Que peut on imaginer comme données internes au système nécessaires au calcul des montants ? Du coup, une fonctionnalité de modification des paramètres du système devient nécessaire, par exemple pour changer le taux de TVA ou de prélèvement pour les ASSEDIC ou la caisse de retraite). Les fonctions ou procédures à définir pour réaliser le système se dessinent également, avec leurs paramètres.

Pour chaque fonctionnalité définie, il faut prévoir une procédure de test. Il s'agit ici de préparer la phase de test en décrivant des procédures et éventuellement des données qui permettront, une fois la programmation effectuée, de vérifier que la fonctionnalité est correctement remplie. Ce travail vous est demandé en phase de conception. Ici, vous pourriez dresser un tableau de validation comme illustré sur le tableau A.1 (la colonne validation sera remplie lorsque la fonctionnalité aura été implantée).

**Réalisation d'une version préliminaire de l'interface :** les différents écrans de l'interface utilisateur doivent être décrits, ainsi que la manière dont ils s'enchaînent en fonction des choix de l'utilisateur.

**Rédaction d'une version préliminaire du manuel utilisateur :** les fonctionnalités doivent être suffisamment bien décrites pour prévoir précisément comment le logiciel pourra être utilisé. Le manuel utilisateur ne vous est pas demandé maintenant, mais pensez lors de la rédaction de votre document

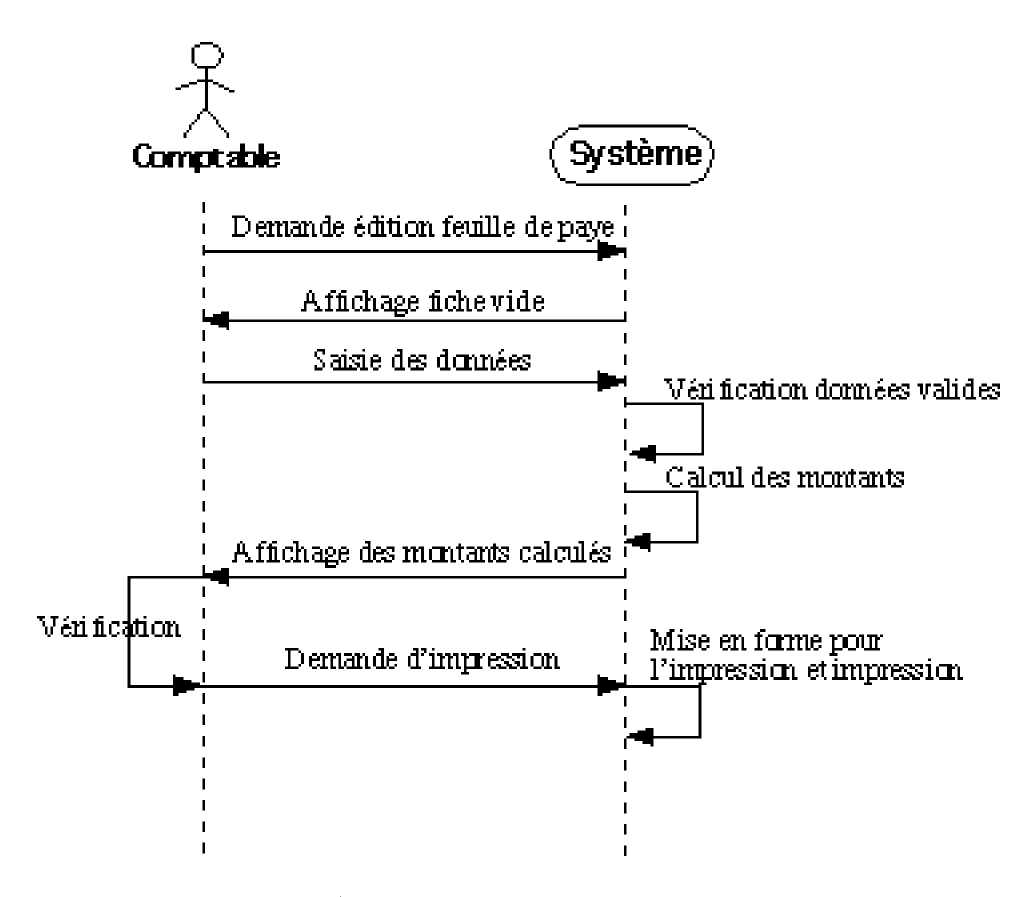



Figure A.1 – Exemple de scénario d'utilisation

de spécification à garder le point de vue de l'utilisateur. Vous pourrez plus facilement vous inspirer de ce premier document lors de la rédaction ultérieure du manuel utilisateur.

## A.3 Conception

L'activité de conception consiste à enrichir la description du logiciel de détails d'implantation afin d'aboutir à une description très proche d'un programme. Elle se déroule en deux étapes : l'étape de conception architecturale et l'étape de conception détaillée. Cette étape peut éventuellement donner lieu à une révision des résultats de l'étape précédente.

### A.3.1 Conception architecturale

L'étape de conception architecturale a pour but de décomposer le logiciel en composants plus simples et indépendants les uns des autres (ces composants sont aussi appelés modules) afin de simplifier l'activité de programmation qui suivra. Cette décomposition doit être faite de sorte que chaque module puisse être réalisé de manière totalement indépendante des autres, par des programmeurs différents. La personne qui, à l'étape de codage, programmera un module doit pouvoir effectuer sa tâche sans avoir besoin de savoir comment les autres modules ont été, sont ou seront programmés.

C'est aussi à cette étape que l'on doit faire les choix contraignant l'implantation :

TABLE A.1 – Tableau de validation

| Fonctionnalité | Valeur nominales, cas particuliers | Validation |
|---|---|---|
| Calcul et édition de feuille de paye | 1. Nom du salarié connu<br>2. Nbre d'heures travaillées : entier compris entre 20 et 39 ; cas particuliers ¡20, ¿39 et ¡ 45 ; si autre chose, message et itération<br>3. Salaire en francs et en euros, avec deux décimales | |
| Modification des paramètres du système | Vérifier les unités et ordre de grandeur des différents paramètres | |

**Configuration matérielle :** sur quel type de machine fonctionnera le logiciel ? Votre logiciel nécessite-t-il la présence de périphériques particuliers (écran graphique, souris, imprimante, scanner, lecteur de CD-ROM, etc.) ?

**Configuration logicielle :** système d'exploitation, compilateur, bibliothèques de fonctions. Pour chacun de ces logiciels, vous devrez préciser le numéro de version.

**Réutilisation de logiciels existants** Dans le cas particulier où vous faites une nouvelle version d'un logiciel déjà opérationnel, il faut choisir entre reprendre et modifier le logiciel ou le refaire complètement.

**Choix de l'environnement de développement :** l'environnement de développement est l'ensemble des logiciels que vous utiliserez pour la programmation (exemple : gcc). Il comprend, au minimum, un compilateur, un éditeur de texte et un débogueur. Ces logiciels peuvent ou non être accessibles à partir d'une unique application.

Pour réaliser le découpage en modules, on part des fonctionnalités qui ont été définies précédemment. Pour chaque fonctionnalité, on définit précisément son interface avec le reste du logiciel, c'est-à-dire l'ensemble des informations dont elle aura besoin pour effectuer sa tâche, celles qu'elle pourra modifier. Puis on la décompose en modules plus simples, dont on précise la fonction réalisée et l'interface avec le logiciel. Ce processus de décomposition est réitéré jusqu'à obtenir des modules aussi simples que possible.

Lors de ce travail vous devez donner un nom le plus explicite possible à chaque module et aux données qu'il utilise ou produit.

Un autre but important de cette décomposition est de mettre en évidence des modules possédant des caractéristiques (fonctionnalité et interface) suffisamment proches pour être fusionnés en un même module plus général réduisant d'autant le travail de programmation. Dans le même ordre d'idée, lorsque des bibliothéques logicielles sont disponibles, la découpe en modules devra, autant que possible, permettre la réutilisation de modules déjà existant dans ces bibliothèques.

En même temps que ce découpage, vous devez faire apparaître les dépendances entre modules : quel module utilise quel module ? Le résultat de ce travail sera exprimé sous forme d'un graphe de dépendances : chaque module est représenté par une boîte et les dépendances sont représentées par des flèches entre ces boîtes.

### A.3.2 Conception détaillée

L'étape de conception détaillée fournit pour chaque composant (ou module) une description de la manière dont les fonctions du composant sont réalisées.

C'est à cette étape que vous devez définir et nommer les structures de données qui seront utilisées dans le programme. Pour les principales structures de données vous devez donner une justification précise des choix effectués.

Lorsque le logiciel utilise des fichiers, c'est à cette étape que vous devez décrire très précisément le format de ces fichiers. Il s'agit de donner tous les détails nécessaires afin que chaque module utilisant ces fichiers puisse être programmé indépendamment des autres.

Pour les modules les plus importants, vous devez de plus donner un algorithme *en franéais* (et non dans un langage de programmation !). A cette étape, il ne s'agit pas de commencer la programmation, mais seulement *de préparer* le travail de programmation en précisant un peu plus la manière dont les fonctionnalités seront réalisées.

Vous devez à ce moment là établir un planning prévisionnel indiquant la liste des tâches à réaliser, la date et le début de chaque tâche, qui la réalise, en veillant à ce que l'enchaînement des tâches dans le temps soit correct : il faut réaliser d'abord les tâches nécessaires à la réalisation d'autres tâches. Et il ne faut pas oublier les tâches de documentation et de préparation de la démonstration.

Vous reprendrez le tableau de validation, en présentant cette fois, pour chaque module la liste des fonctionnalités à implanter, les valeurs d'entrée normales avec les résultats attendus (jeu de test) en précisant les cas particuliers qui peuvent se présenter, et l'état actuel de la validation (pas fait : les tests n'ont pas été passés ; en cours : les tests sont en cours, mais la mise au point n'est pas terminée, il reste des bogues ; OK : les tests ont été passés avec succès). La colonne "validation" sera remplie au fur et à mesure de la mise au point, après le codage, et restera donc vide pour le moment.

## A.4 Codage

C'est seulement à cette étape que commence la programmation. Si les étapes précédentes ont été passées correctement chaque module peut être programmé indépendamment des autres et le travail de programmation pourra être réparti entre les différents participants au projet. Cette phase de programmation doit suivre certaines règles :

- se conformer strictement à la description du module qui a été faite lors de la conception détaillée (fonctionnalités à réaliser et interface avec le reste du logiciel) : tout écart à cette règle peut remettre en cause la réalisation de l'ensemble du projet et réduire à néant tout le travail déjà effectué sur les autres modules.
- adopter des standards de programmation (cf. annexe) pour le nommage des différents objets du langage (fichiers, type, variables, constantes, fonctions, procédures...), pour l'écriture des commentaires, pour la présentation du code (en particulier pour l'indentation), pour le traitement des erreurs (détection systématique des erreurs pouvant être causées par les appels systèmes, format des messages d'erreurs, ...).

### Test et mise au point

Le programme est exécuté sur les données de test et les résultats obtenus sont comparés avec ceux attendus. Lorsque les résultats obtenus différent des résultats attendus, il faut corriger le programme et faire repasser tous les tests. La phase de

test est considérée comme terminée lorsque la version déboguée du logiciel passe tous les tests avec succés.

# Annexe B

# Le mémoire du projet

Lors de la réalisation du projet, chaque étape a donné lieu à la rédaction de documents. Ces différents documents ont du être révisés au fur et à mesure que la réalisation avanéait. Le document final du projet reprend l'essentiel de ces documents dans leur version finale (dans une présentation légèrement différente) plus certaines informations complémentaires.

## B.1 Présentation

Sa présentation est la même que tous les documents intermédiaires. Le mémoire doit être tapé à l'aide d'un traitement de texte. Il doit évidemment comporter un entête ou une première page avec :

– un titre
– le nom de(s) (l')auteur(s) et son(es) courrier électronique
– la matière, la formation et l'année
– le nom de l'enseignant encadrant auquel le document est destiné
– la date à laquelle cette version du mémoire a été terminée

Toutes les pages doivent être numérotées et le mémoire doit comporter une table des matières.

## B.2 Plan du mémoire

Le mémoire proprement dit doit comporter les chapitres suivants :

1. L'énoncé du problème et son analyse approfondie, afin de décrire entièrement le projet à réaliser. Les problèmes (théoriques ou pratiques) sous-jacents à l'énoncé, les fonctionnalités demandées ou nécessaires, les contraintes explicites ou non, l'environnement matériel et logiciel du projet, doivent être détaillés ; les choix effectués doivent être présentés avec leur justification.
2. La conception architecturale du logiciel, afin de définir les constituants du logiciel. Le projet est découpé en modules, chaque module regroupe des procédures ou fonctions. Cette partie du document comprend l'architecture logicielle (la découpe en modules et le graphe de dépendance).
3. La conception détaillée :
   – les paramètres et variables principaux, qui sont nommés et décrits,
   – la liste des en-têtes de procédures ou fonctions classés par module puis ordre alphabétique,
   – la liste des données partagées et leur structure classées par ordre alphabétique,

  - la liste des fichiers comportant les données ou les programmes avec leur structure,
  - l'esquisse des algorithmes cruciaux.

  Pour chaque objet, ajouter un commentaire le décrivant et la page où il est déclaré dans le code. Préciser éventuellement à quoi il sert et dans quels modules, procédures ou fonctions il est utilisé,

4. Un manuel utilisateur en deux parties :
   - le manuel d'installation, qui donne :
     - la liste des fichiers nécessaires pour exécuter le logiciel (librairies, utilitaires, programme source, exécutable) en les commentant ; ne pas oublier de préciser les numéros de version de système ou de compilateur ; tout l'environnement nécessaire à l'exécution doit être décrit ;
     - les fichiers constituant le logiciel et la faéon de les installer sur sa machine.
   - le manuel utilisateur, qui décrit :
     - comment lancer le logiciel ;
     - comment l'utiliser : quelles sont les fonctionnalités possibles, quelles contraintes doivent être respectées, comment se déroule le dialogue utilisateur-logiciel (éventuellement, donner les écrans de l'interface utilisateur), que se passe-t-il en cas d'erreur (liste des messages d'erreur), comment quitter le logiciel (procédure normale et en cas d'erreur).
5. Le planning prévisionnel remis avec la conception détaillée et le planning réel,
6. Le tableau de validation complété,
7. Le cas échéant, les fichiers de données avec lesquels vous avez testé votre logiciel ;
8. Une liste de références bibliographiques. Les références sont les ouvrages (livres, articles, ...) que vous avez utilisés pour l'analyse du problème et la réalisation de votre projet. Ne faites apparaître que les ouvrages que vous avez directement consultés. Une liste de références est une source d'information : elle doit être précise et concise.

L'ensemble de la documentation devra être rendu sous forme papier, avec un fichier contenant code et documentation.

# Annexe C

# Évaluation du projet

L'évaluation du projet sera faite au vu de :

- la qualité technique de la réalisation (en particulier l'adéquation du logiciel au problème posé, mais aussi, le respect des standards de programmation donnés en annexe) ;
- la qualité du mémoire (présentation et contenu) ;
- la qualité de la démonstration du logiciel.
- l'organisation et le travail en groupe.

## Quelques mots sur la préparation de la démonstration du logiciel

C'est lors de la démonstration que la qualité technique de la réalisation sera évaluée.

Préparer une démonstration passe par les étapes suivantes :

1. définir un objectif : ce que l'on veut montrer ;
2. associer un scénario d'utilisation permettant de satisfaire cet objectif ;
3. répéter l'exécution du scénario en situation réelle, c'est-à-dire dans la configuration matérielle et logicielle qui sera celle de la véritable démonstration (en particulier, si vous avez réalisé le projet chez vous, il est très probable que vous devrez y apporter des modifications pour qu'il fonctionne correctement à l'Institut Galilée) et préparer des commentaires oraux.

Vous devez mettre en valeur ce que vous avez fait, et, seulement à la fin, expliquer les erreurs résiduelles et les fonctionnalités non opérationnelles. Les problèmes de gestion de groupe que vous pouvez avoir rencontrés ne doivent pas apparaître lors de la démonstration, ils doivent avoir été évoqués bien avant (au moment où ils se sont produits) avec votre enseignant.

# Annexe D

# Standards de programmation de GNU

Traduction partielle des "GNU Coding Standards" rédigés par Richard Stallman et disponible é `http://gnu.via.ecp.fr/prep/standards_toc.html`

## D.1 Comportement commun à tous les programmes

Ce chapitre décrit comment écrire des logiciels robustes. Il décrit aussi des standards généraux pour les messages d'erreur, l'interface ligne de commande, et la manière dont les bibliothéques de fonctions doivent se comporter.

### D.1.1 Ecrire des programmes robustes

Evitez les limites arbitraires sur la taille des structures de données, y compris la taille des noms de fichiers, des lignes, des fichiers et des symboles. Pour cela, utilisez l'allocation dynamique pour toutes les structures de données. Dans la plupart des utilitaires Unix, les lignes longues sont tronquées sans avertissement. Ce n'est pas toutefois pas acceptable.

Les utilitaires lisant des fichiers ne doivent pas éliminer les caractéres NUL ou tout autre caractère, *y compris ceux dont le code est inférieur à 0177*. Les seules exceptions admises concernent les utilitaires qui sont destinés spécifiquement à servir d'interface à certains types d'imprimantes qui ne peuvent pas gérer ces caractères.

Testez toujours le code d'erreur renvoyé par un appel système. Tout message d'erreur résultant de l'échec d'un appel système doit contenir le texte de l'erreur système (obtenu à l'aide de `perror` ou d'un équivalent) ainsi que le nom du programme dont l'appel système est issu et, le cas échéant, le nom du fichier. Afficher seulement "`cannot open foo.c`" ou "`stat failed`" n'est pas suffisant.

Testez tout appel à `malloc` ou `realloc` pour voir s'il renvoie zéro. Testez `realloc` même si vous diminuez la taille du bloc ; avec un système qui arrondit les tailles de bloc à une puissance de 2, `realloc` peut rendre un bloc différent si vous demandez moins d'espace mémoire.

Sous Unix, `realloc` peut détruire le bloc mémoire s'il renvoie zéro. Le `realloc` de GNU ne présente pas cette bogue : s'il échoue, le bloc original est inchangé. Vous pouvez faire l'hypothése que cette bogue a été corrigée. Si vous désirez exécuter votre programme sous Unix, et souhaitez éviter tout problème dans ce cas, vous pouvez utiliser le `malloc` de GNU.

Vous devez vous attendre à ce que `free` modifie le contenu du bloc qui a été

libéré. Tout ce que vous devez récupérer de ce bloc, doit être récupéré avant d'appeller `free`.

Si `malloc` échoue dans un programme non interactif, faites-en une erreur fatale (i.e. avec terminaison immédiate de l'exécution du programme). Pour un programme interactif (un programme qui lit des commandes de l'utilisateur), il est préférable de faire terminer la commande prématurément et de retourner à la boucle de lecture de commande. Cela permet à l'utilisateur de tuer d'autres processus pour libérer de la mémoire virtuelle et de réessayer la commande aprés.

Utilisez `getopt_long` pour décoder les arguments, à moins que la syntaxe des arguments en rende l'utilisation déraisonnable.

Quand de la mémoire statique (`static`) doit être modifiée lors de l'exécution d'un programme, utilisez du code C explicite pour l'initialiser. Réservez le mécanisme d'initialisation à la déclaration de C pour les données qui ne changeront pas.

Essayez d'éviter les références à des structures de données de bas niveau de Unix (telles que répertoires, `utmp` ou disposition de la mémoire du noyau Unix), car il est peut probable que ce soit compatible avec d'autres implantations d'Unix. A titre d'exemple, si vous avez besoin des noms de tous les fichiers d'un répertoire, utilisez `readdir` ou toute autre interface de haut niveau. Dans ce cas la compatibilité sera assurée par GNU.

Dans les tests d'erreurs qui détecte des conditions "impossibles", contentez vous de faire terminer le programme prématurément. En général, cela ne sert à rien d'afficher quelque message que ce soit. Ces tests indiquent la présence de bogues. Qui que ce soit voulant corriger ces bogues devra lire le code source et utiliser un débogueur. Expliquez donc l'erreur à l'aide de commentaires dans le code source. Les données intéressantes seront dans des variables qui pourront être examinées facilement avec le débogueur.

N'utilisez pas un compteur d'erreurs comme code de retour d'un programme. *Cela ne marche pas* car les valeurs de code de retour des programmes sont limitées à une représentation sur 8 bits (valeurs de 0 à 255). Il est possible qu'une seule exécution d'un programme produise 256 erreurs ; si vous essayez de renvoyer 256 comme code de retour, le processus père verra un code de retour égal à 0, et il supposera que le programme a réussi. Si vous créez des fichiers temporaires, testez la variable d'environnement `TMPDIR` ; si cette variable est définie, utilisez le répertoire spécifié à la place de `/tmp`.

### D.1.2 Bibliothèques de fonctions

Voici maintenant quelques conventions de nommage concernant les bibliothèques de fonctions, afin d'éviter les conflits de nom.

Choisissez pour chaque bibliothèque un préfixe de plus de deux caractéres. Tout nom de fonction ou variable externe doit commencer par ce préfixe. De plus, chaque membre de la bibliothéque ne doit contenir qu'un symbole externe. Cela signifie habituellement que chacun doit être placée dans un fichier source distinct.

Une exception peut-être faite lorsque deux symboles externes sont toujours utilisés ensemble de sorte qu'aucun programme raisonnable n'en utiliserait un sans utiliser l'autre ; dans ce cas, ils peuvent aller tous les deux dans le même fichier.

Les symboles externes qui ne sont pas des points d'entrée documentés pour l'utilisateur (et ne lui sont donc pas destinés) doivent avoir des noms commençant par '_'. Ils doivent eux aussi contenir le préfixe choisi pour la bibliothèque afin de prévenir des conflits avec d'autres bibliothèques. Ils peuvent être placés dans les mêmes fichiers avec des points d'entrée pour l'utilisateur.

### D.1.3 Formattage des messages d'erreur

Les messages d'erreurs des programmes non interactifs doivent ressembler à cela :
*programme* :*nom-fichier-source* :*num-ligne* : *message*
quand il existe un nom de fichier source approprié, ou à cela :
*programme* : *message*
quand il n'y a pas de fichier source pertinent.

Pour un programme interactif (un programme qui lit des commandes d'un terminal), il est préférable de ne pas inclure le nom du programme dans le message d'erreur. L'endroit adéquat pour indiquer quel programme s'exécute, est l'invite (prompt) ou un emplacement spécifiquement réservé dans l'agencement de l'écran. (Quand le même programme s'exécute en prenant son entrée à une autre source qu'un terminal, il n'est pas interactif et devra donc écrire les messages d'erreur dans le style non interactif).

La chaîne de caractéres message ne doit pas commencer par une majuscule quand elle suit un nom de programme ou de fichier. En outre, elle ne doit pas se terminer par un point.

Les messages d'erreur des programmes interactifs ainsi que les autres messages comme les messages d'utilisation (usage), doivent commencer par une majuscule mais ne doivent pas se terminer par un point.

### D.1.4 Standards pour l'interface ligne de commande

Faites en sorte que le comportement d'un logiciel ne dépende pas du nom utilisé pour lancer la commande. Il est parfois utile de faire un lien vers un logiciel en utilisant un nom différent, et cela ne doit pas modifier ce qu'il fait.

Au lieu de cela, utilisez des options d'exécution ou des options de compilation ou bien les deux pour sélectionnez parmi les différents comportements alternatifs.

De la même façon, ne rendez pas le comportement d'un programme dépendant du type de périphérique de sortie avec lequel il est utilisé. L'indépendance par rapport aux périphériques est un principe important de conception des systèmes ; ne le compromettez pas dans le seul but d'éviter à quelqu'un de taper une option par ci par là.

Si vous pensez qu'un comportement est plus adéquat quand la sortie est dirigée vers un terminal, et qu'un autre est plus pertinent lorsque la sortie est dirigée vers un fichier ou un tube (pipe), alors, habituellement, le mieux est de choisir comme comportement par défaut celui qui est le plus pertinent avec un terminal de sortie et de proposer une option pour l'autre comportement.

La compatibilité nécessite que certains programmes soient dépendants du type de périphérique de sortie. Il serait désastreux que `ls` ou `sh` ne le fasse pas de la manière à laquelle tous les utilisateurs s'y attendent. Dans certains cas, nous ajoutons au programme une version alternative préférée ne dépendant pas du type de périphérique de sortie. Par exemple, nous fournissons un programme appellé `dir`, très proche de `ls` excepté que son format de sortie par défaut est toujours multicolonne.

Il est préférable de suivre les principes proposés par POSIX pour les options de la ligne de commande d'un programme. La manière la plus simple de le faire est d'utiliser `getopt` pour les extraire de la ligne de commande. Notez que, normalement, la version GNU de `getopt` autorisera des options n'importe où parmi les arguments à moins que l'argument spécial '`--`' ne soit utilisé. Ce n'est pas ce que spécifie POSIX ; c'est une extension GNU.

Définissez des options à noms longs, équivalentes aux options en une lettre de style Unix. Nous espérons de cette manière rendre GNU plus convivial. C'est facile à mettre en œuvre avec la fonction GNU `getopt_long`.

L'un des avantages des options à noms longs est qu'elles peuvent être consistantes d'un programme à l'autre. Par exemple, les utilisateurs doivent pouvoir s'attendre à ce que l'option ¡¡ verbeuse ¿¿ de tout programme GNU en possédant une s'écrive exactement '`--verbose`'.

Habituellement, il est préférable que les noms de fichier donnés comme arguments ordinaires ne puissent être que des fichiers d'entrée ; tout fichier de sortie devra être spécifié à l'aide d'options (de préférence '`-o`' ou '`--output`'). Même si, pour des raisons de compatibilité, vous autorisez des noms de fichiers de sortie comme arguments ordinaires, essayez d'offrir en plus la possibilité de les spécifier à l'aide d'une option. Cela conduira é une plus grande consistance entre les utilitaires GNU et réduira le nombre de spécificités dont les utilisateurs devront se rappeler.

Tout programme doit supporter deux options standards : '`--version`' et '`--help`'.

### D.1.5 `--version`

Tout programme appelé avec cette option doit afficher sur la sortie standard des informations sur son nom, sa version, son origine et son statut légal, ensuite de quoi le programme se termine normalement. Les autres options et arguments doivent être ignorés dès que cette option est reconnue et le programme ne doit pas assurer son fonctionnement normal. La première ligne est censée pouvoir être analysée facilement par un programme ; le numéro de version est placé juste après le dernier espace. De plus, elle contient le nom canonique de ce programme sous le format suivant :

`GNU Emacs 19.30`

Le nom du programme doit être une chaîne de caractères constante ; *ne la calculez pas* à partir de `argv[0]`. L'idée est de donner le nom standard ou canonique du programme, pas son nom de fichier. Il existe d'autres procédés pour retrouver le nom exact du fichier trouvé par l'intermédiaire de `PATH` pour une commande donnée. Si le programme est une partie annexe d'un package plus important, mentionnez le nom du package entre parenthèses de la manière suivante :

`emacsserver (GNU Emacs) 19.30`

Si le package a un numéro de version différent du numéro de version du programme, vous pouvez mentionner le numéro de version du package juste avant la parenthèse fermante. S'il est nécessaire de mentionner les numéros de version des bibliothèques qui sont distribuées séparément du package qui contient ce programme, vous pouvez le faire en affichant une ligne supplémentaire d'information sur la version pour chaque bibliothèque que vous voulez mentionner. Pour ces lignes, utilisez le même format que pour la première ligne. Ne mentionnez pas toutes les bibliothèques utilisées par le programme "juste par complétude", cela produit un fouillis inutile. Ne mentionnez les numéros de version des bibliothèques que si vous trouvez que c'est très important pour vous pour le déboguage. La ligne suivant la ou les lignes de numéro de version doit être une notice de copyright. Si plusieurs notices de copyright sont nécessaires, mettez-les chacune sur des lignes séparées. Ensuite, doit suivre une brève déclaration sur le fait que le programme est un logiciel libre, et que les utilisateurs sont libres de le copier et le modifier sous certaines conditions. Si le programme est couvert par la GNU GPL, dites le ici. Mentionnez aussi qu'il n'y a aucune garantie dans les limites permises par la loi. Vous pouvez terminer le message affiché par une liste des principaux auteurs du programme, afin de leur attribuer le crédit. Voyez ci-aprés un exemple d'affichage respectant ces règles :

```
GNU Emacs 19.34.5
Copyright (C) 1996 Free Software Foundation, Inc.
GNU Emacs comes with NO WARRANTY,
to the extent permitted by law.
```

```
You may redistribute copies of GNU Emacs
under the terms of the GNU General Public License.
For more information about these matters,
see the files named COPYING.
```

Bien entendu, vous devez adapter cela à votre programme en y mettant les informations correctes concernant l'année, le détenteur du copyright, le nom du programme, le cadre autorisé pour la redistribution et en modifiant les termes employés autant que nécessaire. Dans cette notice de copyright, il n'est nécessaire de mentionner que l'année la plus récente à laquelle des modifications ont été apportées ; il n'est aucunement nécessaire de lister les années concernant les modifications correspondant aux versions précédentes. Vous n'avez pas à mentionner le nom du programme dans ces notices, puisqu'il a déjà été mentionné à la première ligne.

### D.1.6 `--help`

Cette option doit afficher sur la sortie standard une documentation brève sur la manière d'appeler le programme, ensuite de quoi le programme se termine normalement. Les autres options et arguments doivent être ignorés dès que cette option est reconnue et le programme ne doit pas assurer son fonctionnement normal. Vers la fin de cet affichage, il doit y avoir une ligne expliquant où envoyer un mail pour signaler un bogue. Cette ligne doit respecter le format suivant :
`Report bugs to` *`mailing-address`*`.`

### D.1.7 Utilisation de la mémoire

Si vous n'utilisez que quelques mégas de mémoire, ne vous préoccupez pas de faire des efforts pour réduire l'utilisation de la mémoire. Par exemple, si, pour de toutes autres raisons, il n'est pas envisageable en pratique de travailler sur des fichiers de plus de quelques mégaoctets, il est raisonnable de charger entièrement les fichiers d'entrée en mémoire pour travailler dessus.

Cependant, pour des programmes comme `cat` ou `tail`, qui opèrent utilement sur de très grands fichiers, il est important d'éviter d'utiliser une technique qui limiterait artificiellement la taille des fichiers qu'ils pourraient traiter. Si un programme travaille ligne par ligne et doit s'appliquer à des fichiers quelconques fournis par l'utilisateur, il ne doit conserver qu'une ligne en mémoire, car cela n'est pas très difficile à mettre en œuvre et que les utilisateurs voudront être en mesure de travailler sur des fichiers d'entrée plus grands que ceux qui pourraient tenir en mémoire.

Si votre programme crée des structures de données complexes, faites le en mémoire et faites terminer le programme prématurément (fatal error) si `malloc` retourne zéro.

## D.2 Faire le meilleur usage de C

Ce chapitre fournit des conseils sur comment faire le meilleur usage du langage C lors de l'écriture de logiciels GNU.

### D.2.1 Formater votre code source

Il est important de placer en colonne zéro les accolades ouvrantes qui marquent le début du corps d'une fonction C, et d'éviter de placer toute autre accolade ouvrante, parenthése ouvrante ou crochet ouvrant en colonne zéro. Différents outils recherchent les accolades ouvrantes en colonne zéro pour déterminer les débuts de

fonctions C. Ces outils ne fonctionneront pas si vous ne formater pas votre code source de cette manière.

De même, pour les définitions de fonctions, il est important de faire commencer le nom de la fonction en colonne zéro. Cela aide les gens dans la recherche de définitions de fonctions et peut aussi aider certains outils à les reconnaître. Le formatage correct est donc :

```
static char *
concat (s1, s2)        /* Name starts in column zero here */
char *s1, *s2;
{                     /* Open brace in column zero here */
  ...
}
```

ou bien, si vous voulez utiliser ANSI C, formater la définition de cette manière :

```
static char *
concat (char *s1, char *s2)
{
  ...
}
```

En ANSI C, si les arguments ne tiennent pas correctement sur une ligne, scinder la liste des arguments de la manière suivante :

```
int
lots_of_args (int an_integer, long a_long, short a_short,
              double a_double, float a_float)
...
```

Pour le corps de la fonction, nous préférons un code formaté comme ceci :

```
if (x < foo (y, z))
  haha = bar[4] + 5;
else
  {
    while (z)
      {
        haha += foo (z, z);
        z--;
      }
    return ++x + bar ();
  }
```

Il est plus facile de lire un programme quand il y a des espaces avant les parenthèses ouvrantes et après les virgules. Plus particuliérement après les virgules.

Lorsque vous scinder une expression sur plusieurs lignes, scindez la avant un opérateur, pas après. Ceci est la bonne manière de faire :

```
if (foo_this_is_long && bar > win (x, y, z)
    && remaining_condition)
```

Essayez d'éviter d'avoir deux opérateurs de précédences différentes au même niveau d'indentation. Par exemple, n'écrivez pas ceci :

```
mode = (inmode[j] == VOIDmode
        || GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])
        ? outmode[j] : inmode[j]);
```

Au lieu de cela, utilisez des parenthèses supplémentaires de sorte que l'indentation illustre l'imbrication :

```
mode = ((inmode[j] == VOIDmode
         || (GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])))
        ? outmode[j] : inmode[j]);
```

Insérez des parenthèses supplémentaires, de sorte que Emacs indente le code correctement. A titre d'exemple, l'indentation suivante peut sembler correcte si vous la faites à la main, mais Emacs sabotera votre beau travail :

```
v = rup->ru_utime.tv_sec*1000 + rup->ru_utime.tv_usec/1000
    + rup->ru_stime.tv_sec*1000 + rup->ru_stime.tv_usec/1000;
```

L'ajout de quelques parenthèses résoudra le problème :

```
v = (rup->ru_utime.tv_sec*1000 + rup->ru_utime.tv_usec/1000
     + rup->ru_stime.tv_sec*1000 + rup->ru_stime.tv_usec/1000);
```

Formatez les structures de contrôle do-while comme ceci :

```
do
  {
    a = foo (a);
  }
while (a > 0);
```

Utilisez des caractéres de saut de page (control-L) pour séparer votre programme en pages à des endroits logiques (mais pas à l'intérieur d'une fonction). La longueur des pages n'a aucune importance, il n'est pas nécessaire qu'elle corresponde à la longueur d'une page imprimée. Une ligne contenant un saut de page ne doit rien contenir d'autre.

### D.2.2 Commenter votre travail

Tout programme doit commencer par un commentaire disant brièvement à quoi il sert. Exemple : '`fmt - filter for simple filling of text`'.

Dans un programme GNU, écrivez les commentaires en anglais, l'anglais est la langue que pratiquement tous les programmeurs de tous les pays peuvent lire. Si vous n'écrivez pas bien anglais, écrivez vos commentaires en anglais aussi bien que vous le pouvez puis demandez à d'autres personnes de vous aider à les réécrire. Si vous ne pouvez pas écrire de commentaires en anglais, trouvez quelqu'un pour travailler avec vous et traduire vos commentaires en anglais.

Pour chaque fonction, mettez un commentaire décrivant ce que fait la fonction, quelle sorte d'arguments elle prend ainsi que ce que signifient et à quoi sont utilisées les différentes valeurs possibles des arguments. Il n'est pas nécessaire de répéter avec des mots la signification des déclarations C des arguments, si un type C est utilisé de manière habituelle. Si il y a quelque chose de non standard dans son utilisation (comme un argument de type `char *` qui est en réalité l'adresse du second caractère d'une chaîne de caractères et non celle du premier), où que l'une des valeurs possibles ne fonctionnerait pas de la manière à laquelle on peut s'attendre (tel qu'un fonctionnement correct non garantie pour des chaînes de caractères contenants des sauts de lignes), alors vous devez obligatoirement le mentionner.

Expliquez aussi la signification de la valeur retournée, si elle en a une.

Afin que les commandes Emacs sur les phrases fonctionnent, mettez deux espaces après la fin d'une phrase dans vos commentaires. Ecrivez des phrases complètes et

mettez une majuscule au premier mot. En revanche, si un identificateur en minuscules vient en début de phrase, ne lui mettez pas de majuscule ! Changer une des lettres en fait un identificateur différent. Si vous n'aimez pas commencer une phrase par une lettre minuscule, écrivez la phrase autrement.

Le commentaire d'une fonction est plus claire si vous utilisez le nom des arguments pour parler des valeurs des arguments. Le nom de variable lui-même doit être en minuscule, mais écrivez en capitales quand vous parlez de la valeur plutôt que de la variable elle-même. Ainsi, "the inode number NODE_NUM" plutôt que "an inode".

Normalement, redonner le nom de la fonction dans le commentaire qui la précède n'a aucun intérêt, puisque le lecteur peut le voir par lui-même. Il peut néanmoins y avoir une exception lorsque le commentaire est si long que la fonction elle-même pourrait ne pas apparaître à l'écran.

Il doit aussi y avoir un commentaire pour chaque variable statique (`static`), comme :

```
/* Nonzero means truncate lines in the display;
   zero means continue them.  */
int truncate_lines;
```

Tout '`#endif`' doit avoir un commentaire associé, excepté dans le cas de structures conditionnelles courtes (juste quelques lignes) et sans imbrication. Le commentaire doit donner la condition de la structure conditionnelle qui se termine, ainsi que sa signification. Tout '`#else`' doit avoir un commentaire décrivant la condition et la signification du code qui le suit. Par exemple :

```
#ifdef foo
  ...
#else /* not foo */
  ...
#endif /* not foo */
```

en revanche, écrivez les commentaires de cette manière pour un '`#ifndef`' :

```
#ifndef foo
  ...
#else /* foo */
  ...
#endif /* foo */
```

### D.2.3 Utilisation propre des constructions C

Déclarez explicitement tous les arguments des fonctions. Ne les omettez pas sous prétexte que ce sont des `int`.

Les déclarations de fonctions externes et de fonctions apparaissant plus loin dans le fichier source doivent toutes être placées à un endroit vers le début du fichier (quelque part avant le première définition de fonction du fichier) ou bien dans un fichier header (avec suffixe `.h`). Ne mettez pas de déclarations `extern` é l'intérieur des fonctions.

Evitez de réutiliser toujours les mêmes variables locales (avec des noms comme `tmp`) pour des valeurs différentes à l'intérieur d'une même fonction. Pour éviter cela, il est préférable de déclarer une variable locale distincte pour chaque utilisation distincte, et de lui donner un nom porteur de sens. Ce n'est pas uniquement pour rendre les programmes plus simples à comprendre, mais aussi parce que cela facilite l'optimisation faite par les bons compilateurs. Vous pouvez aussi déplacer

la déclaration de chaque variable locale dans le plus petit bloc incluant toutes ses utilisations. Cela rend le programme encore plus propre.

N'utilisez pas de variables locales ou de paramètres masquant des identificateurs globaux.

Ne déclarez pas plusieurs variables en une déclaration s'étendant sur plusieurs lignes. Au lieu de cela, commencez une nouvelle déclaration à chaque ligne. Par exemple, au lieu de ceci :

```
int    foo,
       bar;
```

écrivez soit cela :

```
int foo, bar;
```

soit cela :

```
int foo;
int bar;
```

(Si ce sont des variables globales, chacune doit être précédée d'un commentaire.)

Quand vous avez un `if-else` imbriqué dans un autre `if`, mettez toujours des accolades autour du `if-else`. Ainsi, n'écrivez jamais ceci :

```
if (foo)
  if (bar)
    win ();
  else
    lose ();
```

écrivez toujours cela :

```
if (foo)
  {
    if (bar)
      win ();
    else
      lose ();
  }
```

Si vous avez un `if` imbriqué dans un `else`, ou bien écrivez `else if` sur une ligne comme ceci,

```
if (foo)
  ...
else if (bar)
  ...
```

avec la partie `then` indentée comme la partie `then` précédente, ou bien écrivez le `if` imbriqué entre accolades comme cela :

```
if (foo)
  ...
else
  {
    if (bar)
      ...
  }
```

Ne déclarez pas à la fois une structure et des variables ou des définitions de types dans la même déclaration. Au lieu de cela, déclarez la structure séparément et utilisez la ensuite pour déclarer les variables ou les types.

Essayez d'éviter les affectations à l'intérieur des conditions des `if`. Par exemple, n'écrivez pas ceci :

```
if ((foo = (char *) malloc (sizeof *foo)) == 0)
  fatal ("virtual memory exhausted");
```

écrivez cela à la place :

```
foo = (char *) malloc (sizeof *foo);
if (foo == 0)
  fatal ("virtual memory exhausted");
```

Ne rendez pas le programme inutilement compliqué, juste pour satisfaire les exigences de `lint`. Ne mettez jamais de cast pour un `void`. Un zéro sans cast convient à la perfection pour une constante pointeur nulle, sauf lors d'un appel de fonction à nombre d'arguments variables.

### D.2.4 Choix des identificateurs de variables et de fonctions

Dans un programme, les noms des variables globales et des fonctions constituent une forme de commentaire. En conséquence, plutôt que d'utiliser des noms concis, cherchez plutôt des noms qui donnent des informations utiles sur la signification de la variable ou de la fonction. Dans un programme GNU, de la même manière que pour les commentaires, les identificateurs doivent être en anglais.

Les noms de variables locales peuvent être plus courts, car ils sont utilisés uniquement dans un seul contexte où des commentaires expliquent leur rôle.

Utilisez des caractères underscore pour séparer les mots dans un identificateur, afin que les commandes sur les mots de Emacs puissent leur être appliquées utilement. Tenez vous en aux lettres minuscules ; réservez les majuscules aux constantes définies par macros ou `enum` et aux préfixes qui obéissent à une convention uniforme.

Vous pouvez, par exemple, utiliser des noms comme `ignore_space_change_flag` ; mais n'utilisez pas de nom comme `iCantReadThis`. Les variables indiquant si des options de la ligne de commande ont été spécifiées doivent être nommées d'après la signification de l'option et non d'après la lettre correspondant à l'option. Un commentaire doit indiquer à la fois la signification de l'option et sa lettre. Par exemple :

```
/* Ignore changes in horizontal whitespace (-b).  */
int ignore_space_change_flag;
```

Quand vous voulez définir des noms avec des valeurs constantes entières, utilisez `enum` plutôt que '`#define`'. GDB reconnaît les constantes énumérations.

Utilisez des noms de fichier de 14 caractères ou moins, afin d'éviter des problèmes avec les plus anciennes versions du système System V. Vous pouvez utilisez le programme `doschk` pour tester cela. `doscheck` teste aussi les conflits de nom potentiels si les fichiers étaient chargés sur un système de fichier MS-DOS – libre à vous de vous en soucier ou non.